Parallel & Distributed Computing

UCS645

Assignment - 3

Make File

Name - Kartik Goel
Roll No. - 102303182

System Environment: Linux (Ubuntu), GCC Compiler with OpenMP Support

AIM:

To learn how to write and use Make file.

Question: Calculate the correlation between every pair of input vectors (given m input vectors, each with n numbers).

Using code :-

Using 2-D Array

```cpp
auto correlate_matrix_parallel_2d_array(const std::vector<std::vector<double>>& data) -> std::chrono::duration<double, std::milli> {
        auto start_time{std::chrono::high_resolution_clock::now()};
        if (data.empty()) return std::chrono::duration<double, std::milli>(0);

        size_t rows{data.size()};
        size_t cols{data[0].size()};

        std::vector<double> means(rows);
        std::vector<double> inv_norms(rows);

        #pragma omp parallel for
        for (size_t i = 0; i < rows; ++i) {
                double sum = 0.0;
                for (double val : data[i]) sum += val;
                means[i] = sum / cols;

                double sq_sum = 0.0;
                for (double val : data[i]) {
                        double diff = val - means[i];
                        sq_sum += diff * diff;
                }
                inv_norms[i] = (sq_sum > 0) ? 1.0 / std::sqrt(sq_sum) : 0.0;
        }

        std::vector<std::vector<double>> result(rows, std::vector<double>(rows));

        #pragma omp parallel for schedule(dynamic)
        for (size_t i = 0; i < rows; ++i) {
                result[i][i] = 1.0;
                for (size_t j = i + 1; j < rows; ++j) {
                        double dot_product = 0.0;
                         #pragma omp simd reduction(+:dot_product)
                        for (size_t k = 0; k < cols; ++k) {
                                dot_product += (data[i][k] - means[i]) * (data[j][k] - means[j]);
                        }
                        double corr = dot_product * inv_norms[i] * inv_norms[j];
                        result[i][j] = corr;
                        result[j][i] = corr;
                }
        }

        auto end_time = std::chrono::high_resolution_clock::now();
        return end_time - start_time;
}
```

Using Flat Array

```cpp
auto correlate_matrix_parallel_flat_array(const std::vector<double>& data, size_t rows, size_t cols) -> std::chrono::duration<double, std::milli> {
	auto start_time{std::chrono::high_resolution_clock::now()};

	if (data.size() != rows * cols) {
		throw std::invalid_argument("Data size does not match rows * cols");
	}

	std::vector<double> means(rows);
	std::vector<double> inv_norms(rows);

	#pragma omp parallel for
	for (size_t i = 0; i < rows; ++i) {
		double sum = 0.0;
		size_t row_offset = i * cols; // Calculate offset once

		for (size_t k = 0; k < cols; ++k) {
			sum += data[row_offset + k];
		}
		means[i] = sum / cols;

		double sq_sum = 0.0;
		for (size_t k = 0; k < cols; ++k) {
			double diff = data[row_offset + k] - means[i];
			sq_sum += diff * diff;
		}
		inv_norms[i] = (sq_sum > 0) ? 1.0 / std::sqrt(sq_sum) : 0.0;
	}

	std::vector<double> result(rows * rows);

	#pragma omp parallel for schedule(dynamic)
	for (size_t i = 0; i < rows; ++i) {
		result[i * rows + i] = 1.0;
		size_t i_offset = i * cols;
		for (size_t j = i + 1; j < rows; ++j) {
			size_t j_offset = j * cols;
			double dot_product = 0.0;

			#pragma omp simd reduction(+:dot_product)
			for (size_t k = 0; k < cols; ++k) {
				double val_i = data[i_offset + k] - means[i];
				double val_j = data[j_offset + k] - means[j];
				dot_product += val_i * val_j;
			}

			double corr = dot_product * inv_norms[i] * inv_norms[j];
			result[i * rows + j] = corr;
			result[j * rows + i] = corr;
		}
	}

	auto end_time{std::chrono::high_resolution_clock::now()};
	return end_time - start_time;
}
```

Using Make File:

```
CXX ?= g++
STD = -std=c++23
OMPFLAG = -fopenmp
CXXFLAGS = -O3 -Wall -Wextra -Wpedantic -march=native
TARGET_SEQUENTIAL ?= correlate_matrix_sequential
TARGET_PARALLEL ?= correlate_matrix_parallel
ARGS ?= 1000 1000

ALL_SOURCES = $(wildcard *.cpp)
SOURCES_SEQUENTIAL = $(filter-out main.cpp, $(ALL_SOURCES))
SOURCES_PARALLEL = $(filter-out main_sequential.cpp, $(ALL_SOURCES))
OBJECTS_SEQUENTIAL = $(SOURCES_SEQUENTIAL:.cpp=.o)
OBJECTS_PARALLEL = $(SOURCES_PARALLEL:.cpp=.o)

all: sequential parallel

sequential: $(TARGET_SEQUENTIAL)

$(TARGET_SEQUENTIAL): $(OBJECTS_SEQUENTIAL)
    @echo "Linking $@"
    @$(CXX) $(STD) $(CXXFLAGS) $(OMPFLAG) -o $@ $^

parallel: $(TARGET_PARALLEL)

$(TARGET_PARALLEL): $(OBJECTS_PARALLEL)
    @echo "Linking $@"
    @$(CXX) $(STD) $(CXXFLAGS) $(OMPFLAG) -o $@ $^

%.o: %.cpp
    @echo "Compiling $<"
    @$(CXX) $(STD) $(CXXFLAGS) $(OMPFLAG) -c -o $@ $<

clean:
    @echo "Cleaning..."
    @rm -f *.o $(TARGET_PARALLEL) $(TARGET_SEQUENTIAL)

run: $(TARGET_PARALLEL)
    @./$(TARGET_PARALLEL) $(ARGS)

run-seq: $(TARGET_SEQUENTIAL)
    @./$(TARGET_SEQUENTIAL) $(ARGS)

.PHONY: all sequential parallel clean run run-seq
```

ON EXECUTION:-

1. Default Size 1000 x 1000 (Given in Make File “ARGS ?= 1000 1000”)

Threads	**2D Heap Array Time (ms)**	**2D Heap Array Speed Up**	**Flat Array Time (ms)**	**Flat Array Speed Up**
1	290.36	1.00x	290.36	1.00x
2	185.45	1.57x	232.34	1.25x
4	118.67	2.45x	118.79	2.44x
6	102.34	2.84x	105.86	2.74x
8	97.67	2.97x	95.95	3.03x
10	90.51	3.21x	89.66	3.24x
12	89.12	3.26x	87.41	3.32x
14	89.27	3.25x	84.38	3.44x

16	81.52	3.56x	82.81	3.51x
18	77.02	3.77x	82.66	3.51x
20	80.73	3.60x	76.18	3.81x
22	81.08	3.58x	83.02	3.50x
24	77.82	3.73x	77.71	3.74x

Now Giving Size via command panel:

1. ARGS = “20 20”

Threads	**2D Heap Array Time (ms)**	**2D Heap Array Speed Up**	**Flat Array Time (ms)**	**Flat Array Speed Up**
1	0.01	1.00x	0.01	1.00x
2	0.18	0.07x	0.11	0.12x
4	1.71	0.01x	0.82	0.02x
6	0.38	0.03x	0.15	0.09x

8	0.31	0.04x	0.19	0.07x
10	0.42	0.03x	0.25	0.05x
12	0.49	0.03x	0.22	0.06x
14	0.55	0.02x	0.29	0.04x
16	0.38	0.03x	0.29	0.04x
18	0.41	0.03x	0.29	0.04x
20	0.40	0.03x	0.33	0.04x
22	0.44	0.03x	0.35	0.04x
24	0.42	0.03x	0.28	0.05x

2. ARGS = “100 100”

Threads	**2D Heap Array Time (ms)**	**2D Heap Array Speed Up**	**Flat Array Time (ms)**	**Flat Array Speed Up**
1	0.97	1.00x	0.97	1.00x

2	1.25	0.78x	0.86	1.13x
4	1.77	0.55x	0.55	1.76x
6	0.52	1.88x	0.38	2.54x
8	0.58	1.69x	0.39	2.48x
10	0.48	2.01x	0.42	2.34x
12	0.54	1.81x	0.37	2.66x
14	0.48	2.03x	0.31	3.15x
16	0.48	2.02x	0.36	2.70x
18	0.44	2.20x	0.33	2.92x
20	0.38	2.56x	0.30	3.25x
22	0.36	2.67x	0.40	2.44x
24	0.43	2.29x	0.42	2.32x

3. ARGS = “500 500”

Threads	2D Heap Array Time (ms)	2D Heap Array Speed Up	Flat Array Time (ms)	Flat Array Speed Up
1	93.51	1.00x	93.51	1.00x
2	48.91	1.91x	49.09	1.90x
4	19.69	4.75x	20.64	4.53x
6	11.14	8.40x	13.69	6.83x
8	8.63	10.84x	12.18	7.68x
10	9.24	10.12x	11.29	8.28x
12	8.84	10.57x	10.06	9.29x
14	8.11	11.53x	9.51	9.83x
16	8.01	11.67x	11.24	8.32x

18	8.05	11.62x	11.78	7.94x
20	8.30	11.26x	11.56	8.09x
22	9.04	10.34x	10.07	9.29x
24	9.86	9.49x	10.66	8.77x

4. ARGS = “5000 5000”

Threads	**2D Heap Array Time (ms)**	**2D Heap Array Speed Up**	**Flat Array Time (ms)**	**Flat Array Speed Up**
1	32938.18	1.00x	32938.18	1.00x
2	25008.97	1.32x	27290.56	1.21x
4	15769.14	2.09x	18082.59	1.82x
6	12556.40	2.62x	14963.10	2.20x
8	14487.27	2.27x	14113.19	2.33x
10	13872.16	2.37x	13693.45	2.41x

12	13209.01	2.49x	13053.93	2.52x
14	12733.32	2.59x	13677.95	2.41x
16	11970.81	2.75x	12196.55	2.70x
18	11876.10	2.77x	11951.39	2.76x
20	11954.72	2.76x	11932.35	2.76x
22	11925.08	2.76x	11812.68	2.79x
24	12136.48	2.71x	12301.64	2.68x

INFERENCE:-

For the 20 x 20 matrix, the speedup is actually a slowdown. We see values like 0.01, which means the parallel version is 100 times slower than the simple sequential one.

This happens because the computer takes a small amount of time to create threads and assign work to them. For a tiny 20 x 20 matrix, the math is finished in a fraction of a millisecond. If the time it takes to manage the threads is longer than the time it takes to do the math, adding more threads just adds more weight without any benefit.

The 500 x 500 matrix shows the most impressive results, reaching a peak speedup of 11.67.

At this size, the amount of work is large enough to keep many threads busy, but the data is likely small enough to fit inside the CPU's cache which is a very fast, internal

memory. Because the CPU doesn't have to wait for the slower main RAM to send data, the speed scales beautifully as we add more threads.

As we move to the 5000 x 5000 matrix, the speedup starts to level off, dropping from the high peaks of the smaller matrices down to around 2.7.

This occurs because we have hit a physical limit called memory bandwidth. Even though we have 24 threads ready to calculate, they all have to share the same path to the system's RAM. At this scale, the CPU is essentially sitting idle while waiting for the RAM to deliver the next set of numbers.

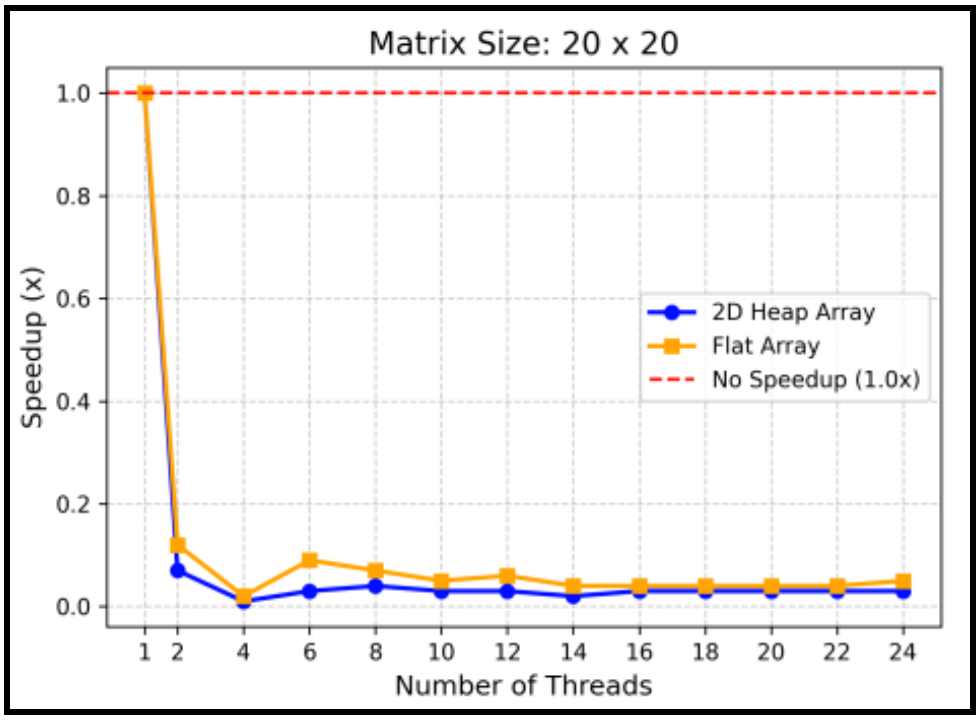
The data compares :-

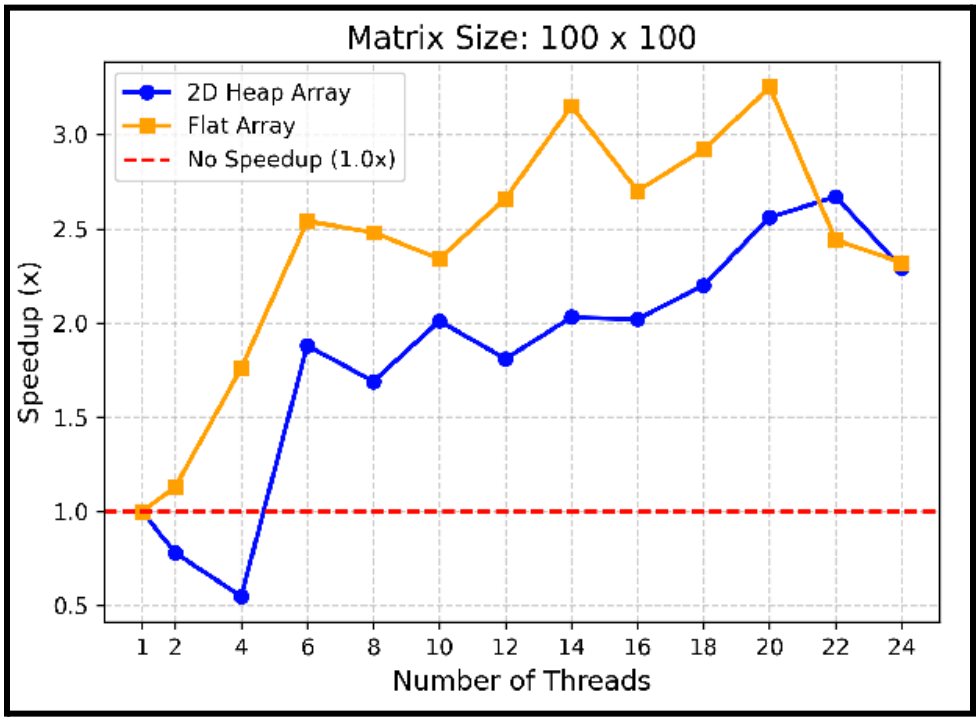
1. 2D array implementation (arrays of arrays) can sometimes be scattered in different parts of the memory.
2. Flat Array implementation (a single long list) usually performs more consistently because it is more cache-friendly.

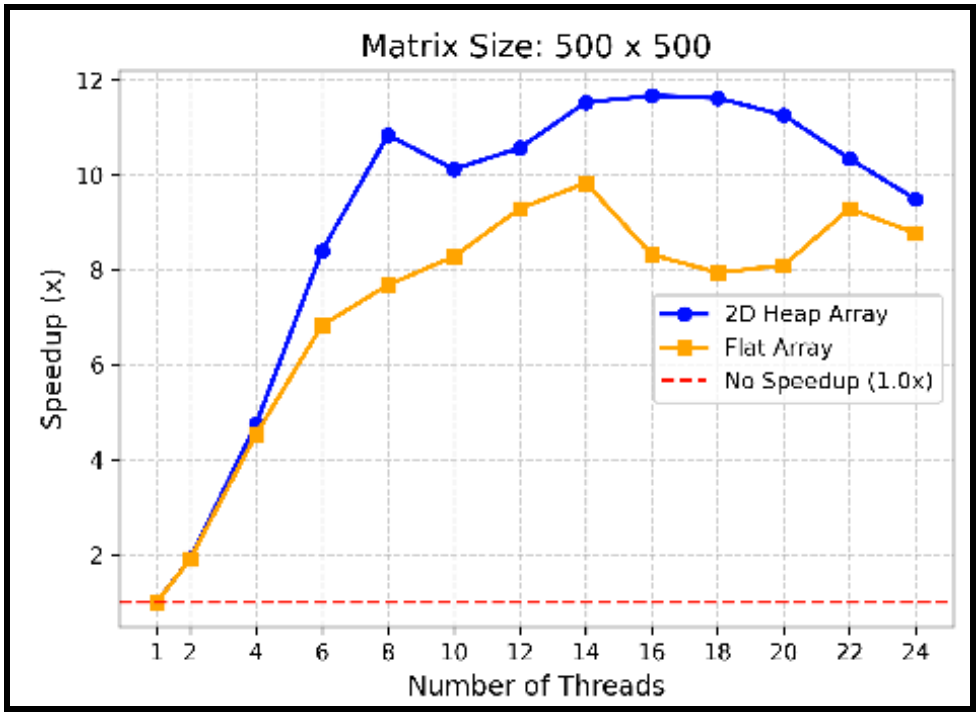
When data is flat, the CPU can predict what we need next and pre-load it. In the 5000 x 5000 results, we can see that both methods eventually converge because the sheer volume of data overwhelms any small advantages in how the memory is organized.

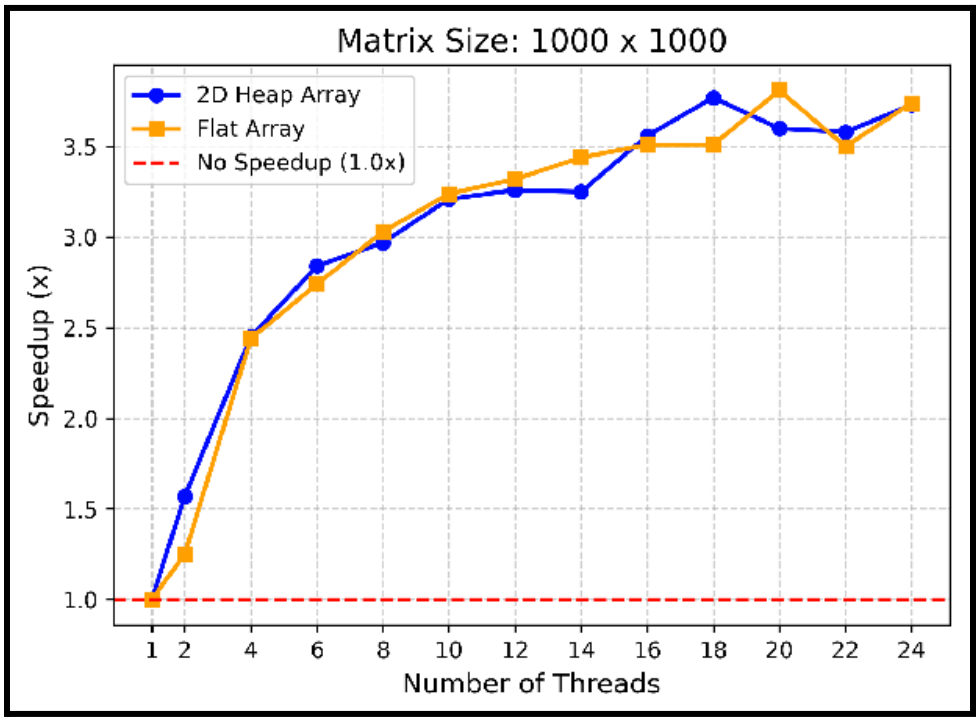
The Makefile is doing a lot of the heavy lifting behind the scenes.

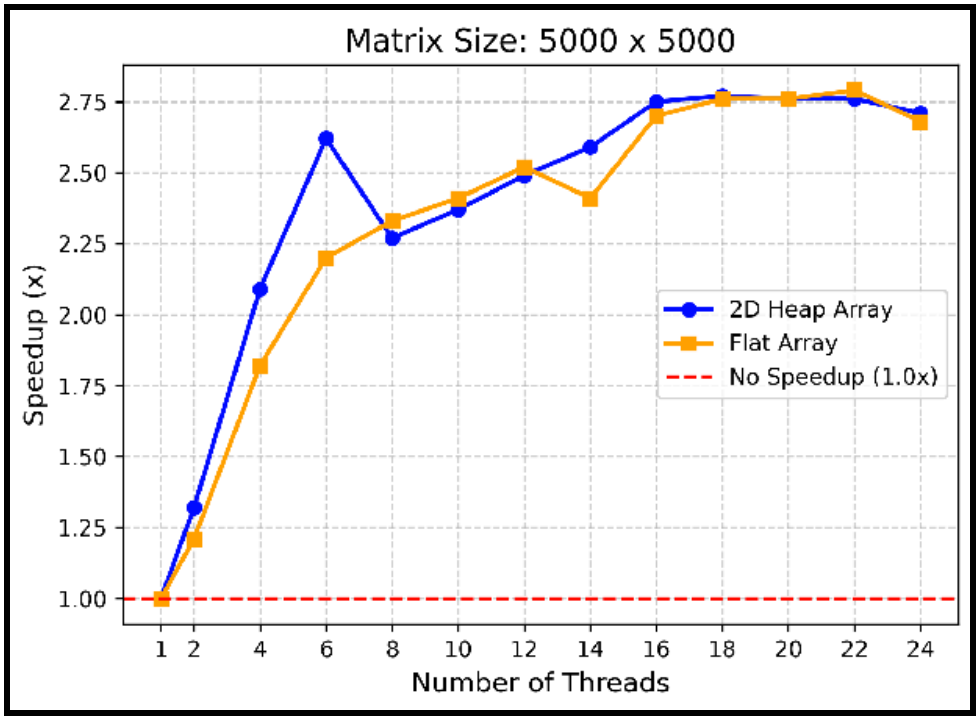
1. The -O3 Flag: This tells the compiler to go into high-gear and rewrite Wer code to be as fast as possible, such as unrolling loops.
2. The -march=native Flag: This is a crucial setting. it tells the compiler to look at the specific CPU and use its specialized hardware instructions (like AVX). This allows the CPU to process multiple numbers at the exact same time , which is why we see such high speedups in the mid-sized matrices.
3. The -fopenmp Flag: This is the key that unlocks the ability to use multiple cores at once. Without this, the code would only ever use one thread, regardless of the matrix size.


Matrix Size: 20 x 20
Speedup (x)
Number of Threads
2D Heap Array
Flat Array
No Speedup (1.0x)
1.0
0.8
0.6
0.4
0.2
0.0
1
2
4
6
8
10
12
14
16
18
20
22
24


Matrix Size: 100 x 100
Speedup (x)
Number of Threads
2D Heap Array
Flat Array
No Speedup (1.0x)
3.0
2.5
2.0
1.5
1.0
0.5
1
2
4
6
8
10
12
14
16
18
20
22
24


Matrix Size: 500 x 500
Speedup (x)
Number of Threads
2D Heap Array
Flat Array
No Speedup (1.0x)
12
10
8
6
4
2
1
2
4
6
8
10
12
14
16
18
20
22
24


Matrix Size: 1000 x 1000
Speedup (x)
Number of Threads
2D Heap Array
Flat Array
No Speedup (1.0x)
3.5
3.0
2.5
2.0
1.5
1.0
1
2
4
6
8
10
12
14
16
18
20
22
24


Matrix Size: 5000 x 5000
Speedup (x)
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
1
2
4
6
8
10
12
14
16
18
20
22
24
Number of Threads
2D Heap Array
Flat Array
No Speedup (1.0x)

Perfstat Inference :-

1. Sequential Execution:

```
Performance counter stats for './correlate_matrix_sequential 1000 1000':

      249,719,980      task-clock                       #    1.100 CPUs utilized
               76      context-switches                 #  304.341 /sec
               42      cpu-migrations                   #  168.188 /sec
            6,132      page-faults                      #   24.556 K/sec
    1,009,971,077      cpu_atom/instructions/           #    3.17  insn per cycle              (2.44%)
    2,087,233,577      cpu_core/instructions/           #    1.95  insn per cycle              (96.28%)
      318,505,070      cpu_atom/cycles/                 #    1.275 GHz                         (3.72%)
    1,072,855,627      cpu_core/cycles/                 #    4.296 GHz                         (96.28%)
      106,307,837      cpu_atom/branches/               #  425.708 M/sec                       (3.72%)
      148,363,230      cpu_core/branches/               #  594.118 M/sec                       (96.28%)
          300,528      cpu_atom/branch-misses/          #    0.28% of all branches             (3.72%)
          567,101      cpu_core/branch-misses/          #    0.38% of all branches             (96.28%)
#     64.4 %  tma_backend_bound
                                                  #      0.8 %  tma_bad_speculation
                                                  #      1.6 %  tma_frontend_bound
                                                  #     33.2 %  tma_retiring             (96.28%)
#     12.1 %  tma_bad_speculation
                                                  #     68.2 %  tma_retiring             (3.72%)
#      0.0 %  tma_backend_bound
                                                  #     19.7 %  tma_frontend_bound       (3.72%)

      0.226957115 seconds time elapsed

      0.230111000 seconds user
      0.023213000 seconds sys
```

1. Resource Utilization:-

Strictly Sequential: With only 1.100 CPUs utilized, this application is entirely single-threaded. The task-clock time (249.71 ms) almost perfectly mirrors the elapsed physical time (226.95 ms), meaning the program is restricted to a single core and is ignoring the rest of the multi-core system.

P-Core Dominance: The operating system effectively routed this single-threaded workload to the fast Performance-cores, which handled 96.28% of the execution time while boosting to an impressive 4.296 GHz. The Efficiency (Atom) cores saw negligible use (3.72%), likely just handling brief background OS tasks or thread initialization.

2. Instruction Efficiency:-

P-Core Performance: The Performance cores achieved a solid IPC (Instructions Per Cycle) of 1.95, successfully retiring 33.2% of their operations without stalling.

Highly Predictable Logic: Branch prediction is near-perfect, with miss rates at a tiny 0.38% on P-cores and 0.28% on E-cores. Because matrix operations rely on highly

repetitive, standard nested for loops, the CPU has absolutely no trouble guessing the control flow.

3. Primary Bottlenecks:-

Memory Starvation (64.4% Backend Bound on P-Cores): Despite running at high clock speeds and predicting branches perfectly, the P-cores spend nearly two-thirds of their execution pipeline completely stalled in the backend. Matrix correlation requires continuously fetching large rows and columns of data. The CPU's arithmetic units are processing the math much faster than the RAM/cache can supply the data, leaving the processor starved and waiting.

The Single-Thread Limit: Outside of the memory stalls, the biggest absolute bottleneck to the execution speed is the lack of parallelization. Because it is only utilizing 1 CPU, the overall performance is hard-capped by the single-core clock speed (4.296 GHz) and cannot scale, no matter how much compute power the rest of the machine has.

2. Parallel Execution:

```
Performance counter stats for './correlate_matrix_parallel 1000 1000':

   12,415,210,763      task-clock                       #    7.647 CPUs utilized
            2,087      context-switches                 #  168.100 /sec
              475      cpu-migrations                   #   38.260 /sec
           52,669      page-faults                      #    4.242 K/sec
   67,639,508,186      cpu_atom/instructions/           #    1.79  insn per cycle              (30.82%)
  129,885,091,795      cpu_core/instructions/           #    2.64  insn per cycle              (64.73%)
   37,842,066,507      cpu_atom/cycles/                 #    3.048 GHz                         (30.88%)
   49,181,901,950      cpu_core/cycles/                 #    3.961 GHz                         (64.73%)
    2,020,643,931      cpu_atom/branches/               #  162.756 M/sec                       (30.89%)
    3,965,426,557      cpu_core/branches/               #  319.401 M/sec                       (64.73%)
        8,695,253      cpu_atom/branch-misses/          #    0.43% of all branches             (30.83%)
       15,872,929      cpu_core/branch-misses/          #    0.40% of all branches             (64.73%)
#     25.2 %  tma_backend_bound
                                                  #      1.1 %  tma_bad_speculation
                                                  #      7.0 %  tma_frontend_bound
                                                  #     66.7 %  tma_retiring             (64.73%)
#      1.7 %  tma_bad_speculation
                                                  #     50.0 %  tma_retiring             (30.85%)
#     47.0 %  tma_backend_bound
                                                  #      1.3 %  tma_frontend_bound       (30.88%)

      1.623454612 seconds time elapsed

     12.286495000 seconds user
      0.143203000 seconds sys
```

1. Resource Utilization:-

Parallel Efficiency: With 7.647 CPUs utilized, the system is successfully parallelizing the workload. It condensed over 12.4 seconds of total computational work (task-clock) into just 1.62 seconds of real-world elapsed time.

Hybrid Execution: The workload is effectively distributed across the hybrid architecture. The Performance-cores are boosting to ~3.96 GHz and handling the majority of the heavy lifting (taking up ~64.7% of the total cycles), while the Efficiency-cores are actively contributing at ~3.05 GHz.

2. Instruction Efficiency:-

Exceptional P-Core Performance: The Performance cores achieved an outstanding IPC (Instructions Per Cycle) of 2.64. Even more impressive is the 66.7% retiring rate. This means exactly what we want to see in a compute heavy task: the fast cores are churning through the math with very few stalls.

Solid E-Core Contribution: The Efficiency cores also performed remarkably well, maintaining an IPC of 1.79 and successfully retiring 50.0% of their instructions.

Perfect Branch Prediction: The predictable nature of nested for loops means the CPU rarely guesses the wrong path. Branch misses are practically non-existent at 0.40% (P-cores) and 0.43% (E-cores).

3. Primary Bottlenecks:-

Memory Starvation (25.2% Backend Bound on P-Cores): This is the biggest victory of the parallel implementation. By parallelizing the code we have drastically reduced memory starvation. The execution units are finally getting the data fed to them fast enough to keep busy.

Moderate E-Core Backend Bound (47.0%): Efficiency cores are spending about half their time waiting on memory, which is expected since E-cores typically have smaller localized caches and less memory bandwidth than P-cores.

3. Comparison between sequential and parallel:

1. Resource Utilization & Core Scaling

Hardware Saturation: The sequential code behaved exactly as expected, utilizing 1.100 CPUs and relying almost entirely on a single Performance-core boosting to 4.296 GHz. The parallel version successfully distributed the workload, utilizing 7.647 CPUs and effectively splitting the operations between P-cores (64.7%) and E-cores (30.8%).

2. Pipeline Efficiency:

Memory starvation: The parallel code is structurally much better optimized for the hardware cache. The sequential version was severely memory-starved, suffering a 64.4% Backend Bound stall rate on the P-cores. The parallel version slashed this memory bottleneck down to just 25.2%.

Instruction Throughput: Because it wasn't waiting on memory, the parallel version saw its P-core retiring rate double (from 33.2% to 66.7%) and its IPC jump from a respectable 1.95 to an outstanding 2.64.